Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors

A. L. NARASIMHA REDDY AND PRITHVIRAJ BANERJEE, MEMBER, IEEE

Abstract—**In this paper, we look at several issues concerning the design of an I/O system for a multiprocessor such as a hypercube. We propose a methodology for connecting the I/O processors to such a system for efficient I/O access. An analysis is presented to see the effect of I/O communication on the network of the multiprocessor. We evaluate different disk organizations that can be employed within such a system to see which organization has a better performance. Then we consider the problem of mapping specific data structures such as matrices onto the disks such that the data can be accessed efficiently.**

Index Terms—**Data distribution, hypercubes, I/O embedding, multiple-disk systems, multiprocessors, performance evaluation, scientific applications.**

I. Introduction

A HYPERCUBE multiprocessor consists of 2^n processors interconnected in an n-dimensional binary cube topology. Each processor in such a system has its own memory and the processors communicate by message passing. Hypercubes are known to have several useful properties, namely a high degree of connectivity, fault tolerance, low diameter, etc. In addition, different interconnections such as a linear array, mesh are known to map easily onto the hypercube interconnection. Due to these advantages, several hypercube systems have been built recently [1]–[3]. Many parallel computations have been developed for the hypercube topology [4]. One issue that has not been addressed effectively in the past is the support of efficient I/O operations in multiprocessors. If the I/O operations are not speeded up, even if we solve the problem considerably faster using a multiprocessor, the speed of the solution will eventually be determined by the serial I/O bottleneck. The importance of balancing I/O bandwidth and computational power has been pointed out in [5] where it is shown that for some applications, the I/O problem cannot be alleviated by simply adding more memory at the processors.

I/O processors are used for transferring data between the hypercube nodes and the outside world and the host. This is to be distinguished from the hardware that is required for communication between the processors. I/O communication is required to distribute data and code to the processors before the computation and to collect the results after the computation has been completed. Each processor in the system is connected to one or more I/O processors and the I/O processor(s) handles all the data transfer between that processor and the outside world.

The Intel iPSC/1 system uses I/O hardware within each processor for I/O communication using the ethernet protocol [2]. A single host serves as the I/O handler as well and thus creating a serial I/O bottleneck. The recently announced I/O system for iPSC/2 uses multiple disks and data declustering to improve the I/O performance [6]. In this system, multiple disks are used and the file is declustered among the different disks to improve the concurrent access to a file. In the NCUBE system, an I/O processor is connected to a subcube of eight processors and the I/O processors are themselves interconnected partially [1]. In Section II, we describe a way of connecting I/O processors to the hypercube system that is more efficient in a number of ways as explained later. Our method uses the system links efficiently for both the I/O and processor-to-processor communication. The proposed method requires no explicit processor-to-I/O processor links. Our method can be directly used in the hypercubes with minor modifications to the architecture.

Some of the alternatives for improving the response time of a disk system include disk striping [7], and disk synchronization [8]. We evaluate these alternatives and combinations of them to find out which organization has a better performance. The multiprocessor network could affect the I/O performance due to traffic patterns within the network and hence there is a need to evaluate such disk organizations within a multiprocessor environment. The I/O traffic may in turn affect the communication in the network, hence there is a need to see how the I/O transfers interact with the message communication. We evaluate the disk organizations using two workloads: 1) file/transaction system workload and 2) scientific applications workload.

The results obtained in evaluating the disk organizations motivated us to study different ways of breaking up a file to be stored among different disks. Since most of the scientific applications need matrix computations, matrices warrant special attention. We present a method for storing the matrices on the different disks such that the I/O requirements of the matrix applications can be efficiently satisfied. We show that

Manuscript received June 9, 1989; revised October 4, 1989. This work was supported in part by an IBM Graduate Fellowship and in part by a National Science Foundation Presidential Young Investigator Award under Contract NSF MIP 86-57563 PYI.

The authors are with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

IEEE Log Number 8934116.

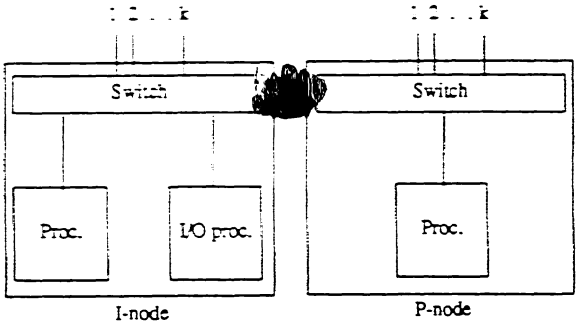


Fig. 1. Architecture of the modes.

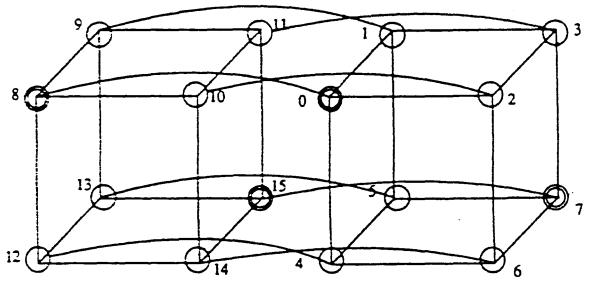


Fig. 2. An I/O embedding in a 4-cube.

some of the earlier ideas on organizing hierarchical memories can be applied in a distributed memory environment also.

The rest of the paper is organized as follows. In Section II, we discuss a method of connecting I/O processors to the hypercube system. In Section III, an analysis is presented on the effects on communication as a result of connecting a disk system to the hypercube. In Section IV, we present simulation results of a system evaluation. In Section V, we discuss how to distribute specific data structures such as matrices onto the disk system for efficient access during the problem execution. Section VI summarizes the results obtained and points out the directions for future research.

II. I/O Embedding in Hypercubes

In this section, we look at the problem of connecting the I/O processors to a hypercube multiprocessor system. If each processor in the system is adjacent to an I/O processor, then the communication between the I/O processor and the processor can be carried out efficiently. We propose a method of interconnecting the I/O processors and the multiprocessor that satisfies this condition.

The communication that takes place among the processors during the problem solution is called *message communication* and the communication required for I/O transfer is called *I/O communication*. We define the problem of connecting the I/O processors to the rest of the system as the *I/O embedding* problem. If each processor in the system is adjacent to at least one I/O processor, then the I/O transfers can be done efficiently. The hypercube system with an embedded I/O system consists of two types of nodes, *P*-nodes and *I*-nodes as shown in Fig. 1. A *P-node* contains a processor and an *I-node* contains a processor and an I/O processor that handles the data transfer to and from the disks. A similar idea of using I/O nodes is employed in the hypernet architecture [9]. However, the main aim of that architecture is to develop networks with a constant degree and the I/O embedding approach has not been explored. In our architecture, a processor is included in the *I*-node so that the hypercube structure is maintained for data processing. The I/O processor in each *I*-node in an n-dimensional hypercube can serve as an I/O processor to its neighboring n processors in the network and the processor in its node. Hence, if sufficient number of *I*-nodes are embedded in the multiprocessor network, then each processor can be made adjacent to an I/O processor.

A *perfect I/O embedding* is an I/O embedding where each processor in the system is adjacent to exactly one I/O processor. A system having a perfect embedding has the fewest possible I/O nodes such that each processor is adjacent to one I/O processor.

Theorem 1: A perfect embedding exists in a k-dimensional hypercube if and only if $k = 2^l - 1$ for some integer l.

Proof: Each *I*-node serves as an I/O processor for $(k+1)$ nodes in the system. Hence, if a perfect embedding exists, there must be exactly $n/(k+1)$ *I*-nodes in the system, i.e., $n/(k+1)$ must be an integer where $n = 2^k$ is the number of nodes in the system. Since 2^k has only factors of the form 2^l, $(k+1) = 2^l$, for some l.

To prove that $k = 2^l - 1$ is sufficient, consider two *I*-nodes a and b in a perfect embedding. Nodes a and b cannot be adjacent. If they are, the processors in nodes a and b are each adjacent to two I/O processors. Similarly, neighbors of a cannot be neighbors of b. Hence, any two *I*-nodes in a perfect embedding are at a Hamming distance of three or greater from each other. Hamming codes give a characterization of such a situation since each Hamming code is at a distance of three or greater from each other. Hamming codes are perfect codes [10] when the code space has a dimension of the form $k = 2^l - 1$. In such a code space, each noncode word is adjacent to exactly one code word. The hypercube can be seen as a k-dimensional space and we can place *I*-nodes in the code-word locations to get the desired perfect I/O embedding. Hence, we can always find a perfect I/O embedding when $k = 2^l - 1$. □

Since all the sizes of the hypercube do not have perfect embeddings, we use smaller-sized perfectly embedded hypercubes to build larger systems. Fig. 2 shows a four-cube with an I/O embedding. It is noted that each processor in the 4-cube is adjacent to at least one I/O processor. The two three-cubes (0-7) and (8-15) in the four-cube are perfectly embedded. Some of the advantages of this approach are [11]: 1) the links connecting the *P*-nodes and *I*-nodes are used efficiently both for message communication and I/O transfer. Since in most of the applications, the I/O and message communication are done at different instants of time, the links can be shared for both the purposes. Otherwise, an extra communication link will be required. 2) Each processor in the system is adjacent to an I/O node so that the processors can communicate with the I/O nodes efficiently. 3) As the system size increases, it is possible for each processor to be adjacent to more than one I/O node, unlike in the NCUBE and Intel systems. 4) It is possible to build larger size hypercubes with the same number of ports for communication.

In our approach, the larger size hypercubes are built from

perfectly embedded smaller-sized cubes. It is possible to interconnect these smaller cubes in various different ways to obtain different properties. Consider, for example, building a larger system from three-dimensional hypercubes. The *I*-nodes are located along the diagonal of a perfectly embedded three-cube. Depending on the orientation in which these different subcubes are connected, the larger system can have different properties. If the subcubes are oriented along the same direction as shown in Fig. 2, the *I*-nodes are connected to each other in a hypercube topology of dimension $n-2$ where n is the dimension of the hypercube system. These connections can be quite efficiently utilized for transferring data between different I/O processors.

If the subcubes are oriented among different directions, it is possible to obtain an embedding where each processor in the system is adjacent to two or more *I*-nodes in the system. This approach is useful when it is necessary to connect each processor to two different sets of I/O processors, for example, a graphics processor and an I/O handler. Consider building a hypercube system from perfectly embedded three-cubes. Let the subcubes of this system be numbered $S_1, S_2, \cdots, S_m$. Let the nodes in the system be numbered by S_i, the ith subcube to which the node belongs and one of $0, 1, \cdots 7$, the node number within the subcube. A subcube is said to have an *i*-specified embedding if the node i in the subcube is an *I*-node. If the subcube S_j has j *mod* 8-specified embedding, then an n-cube has an embedding such that each node is adjacent to l *I*-nodes where $n = 2^l - 1$. For more details, please refer to [11].

The embedding ideas can be extended to *distance-k* embeddings. A system is said to have a *distance-k* embedding if each processor is within a distance of k from an I/O processor. A perfect embedding is then a distance-1 embedding. Bose–Chaudary–Hocquenghem (BCH) codes [10] give a characterization of such embeddings. BCH codes are generalized Hamming codes and can be used to generate distance-k embeddings, for $k \geq 2$. Distance-k embeddings are suitable for very large systems or when the I/O resources are very scarce. A similar approach is adopted by [12] for distributing resources in a hypercube.

In most of the applications the I/O and message communication phases do not overlap and hence utilizing the system links for both the purposes is feasible. The messages can be given higher priority to reduce any performance degradation. Simulation results are presented in later sections to show that the performance degradation is not significant.

III. Communication Issues

In the proposed architecture, I/O messages also use the system links. In regularly decomposable problems where I/O and message-communication do not overlap in time, this does not lead to any congestion of links. In this section, we derive a worst-case analysis of the communication network to see how sharing the links may affect the system performance. We show that under moderate communication loads, using system links for I/O traffic does not degrade performance to a great extent.

To understand the effects of using both I/O messages and processor-to-processor messages on the same links, we studied the way the communication takes place in the hypercube system. Each node contains a $k+1$ by $k-1$ switch as shown in Fig. 3 where k is the dimension of the hypercube. One of the input ports and output ports are used for the node source and node sink ports. The switch output ports are numbered from 0 to n where each output port $i = n$ connects the node to a neighbor along the i-dimension and the output port n routes the incoming messages to the processor node. For convenience, we numbered the input ports from -1 to $n-1$ where input port $i \neq -1$ receives messages from the neighbor along the i-dimension and input port -1 denotes the messages being sent out by the processor node. We assume that the system uses *e*-cube routing. Most current generation hypercube systems use *e*-cube routing since it is shown to be deadlock-free [13]. In *e*-cube routing, a message from a source to a sink is routed along increasing dimensions of the hypercube. For example let the source be $s_{n-1}s_{n-2}\cdots s_0$ and the destination be $d_{n-1}d_{n-2}\cdots d_0$ with the difference vector between the two addresses being $e_{n-1}e_{n-2}\cdots e_0$. The message is sent first along the dimension i where i is the least integer with $e_i = 0$. The message is then sent along the next higher dimension with $e_j \neq 0$ and so on till it reaches the destination. In Fig. 3 a message arriving at an input port i can only be sent along a link j at the output for $j > i$. This restriction is a direct result of *e*-cube routing. This results in an asymmetric traffic pattern within the switch. Various results studying the symmetric traffic pattern analysis have appeared in literature [14]–[16].

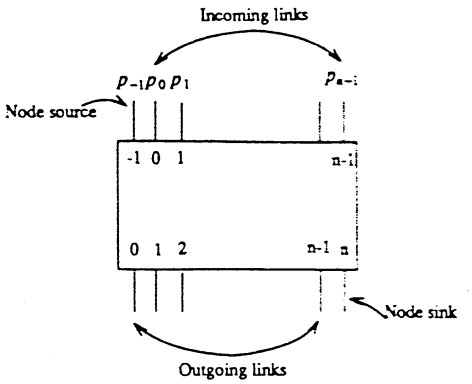


Fig. 3. Routing switch at a node.

Let $p_{-1}, p_0, \cdots, p_n$ be the probabilities that a message arrives at input ports $-1, 0, \cdots, n$ of a routing switch. Let p_{ij} for $-1 \leq i < n$ and $j > i$, denote the probability that a message arriving at input port i is directed to output port j. Then the probability of a message being received by the node can be derived as follows: the probability of not receiving a message at a given node from input port i is given by $1 - p_i p_{in}$. Then the probability that a message is not received from any of the input ports is given by

$$\prod_{i=-1}^{i=n-1} (1 - p_i p_{in})$$

or the probability of receiving a message is given by

$$1 - \prod_{i=-1}^{i=n-1} (1 - p_i p_{in}).$$

Since the system is symmetric and uniform about every node, the probabilities p_i and p_{ij} at each switch should be the same for any message distribution, as long as each node has the same message distribution.

If the probabilities p_i and p_{ij} can be derived, the problem can be solved as above. We derive the necessary probabilities for the case of uniform message routing below. A similar method can be applied for other cases of routing distributions.

Each 0-dimensional link is used by exactly two processors to send or receive messages. Each even numbered processor uses its 0-dimensional link to send or receive a message from the odd numbered processors. Similarly, each processor uses exactly two one-dimensional links for sending or receiving messages. In general, each processor uses 2^i i-dimensional links for routing messages.

Lemma 1: The probability that a particular i-dimensional link is used by a processor for sending a message is $1/2^{i+1}$ when each processor is equally likely to send messages to every processor including itself.

Proof: The probability of using an i-dimensional link for a message routing is given by the probability that the source and destination addresses differ in the ith bit. In case of uniform message routing this probability is 1/2. Since each processor uses exactly 2^i i-dimensional links, the probability that a particular i-dimensional link is used for routing a message is given by $1/2^{i+1}$. □

Lemma 2: The probability that a particular i-dimensional link is used for sending a message is 1/2 when every processor is sending messages randomly to other processors with equal probability.

Proof: Each i-dimensional link is shared by 2^i processors for message routing. Since the probability that a particular processor uses this link for message routing is $2^{1/(i+1)}$, the probability that this link is used by one of the processors is $2^i * 1/2^{i+1} = 1/2$. □

If each processor generates a message with a probability of p during a cycle, then the expected probability that a link (in one direction) is used during a cycle is $p/2$, if no collisions occur during the message routing. If we assume that the messages are generated infrequently, $p/2$ gives an approximation to the above probability even under collisions.

Lemma 3: The probability p_{ij}, for $-1 \leq i < n$, $j > i$, $j \neq n = 1/2^{j-i}$ and $p_{in} = p_{in-1}$, for all i.

Proof: The probability that a message arriving at input port i is directed to j-dimensional link at the output is the probability that the source and destination addresses do not differ in the $i+1, i+2 \cdots j-1$ bits and they differ in the jth bit and this probability is exactly $1/2^{j-i}$. The probability that a given message is directed to the node is the probability that the message is not sent to any of the other possible output ports. Hence, $p_{in} = 1 - \sum_{j=0}^{n-1} p_{ij} = 1/2^{n-i-1}$. □

The above lemmas give the traffic distribution at a switch as shown in Fig. 3. The probability that output port n does not receive a message from input port i is given by

$$\frac{p}{2}\left(1 - \frac{1}{2^{n-i-1}}\right) + \left(1 - \frac{p}{2}\right) = 1 - \frac{p}{2^{n-i}}, i \neq -1 \text{ and}$$

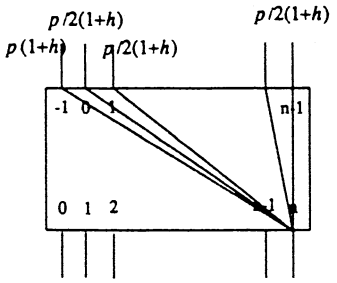


Fig. 4. I/O traffic at a routing switch.

$$p\left(1 - \frac{1}{2^n}\right) + (1-p) = 1 - \frac{p}{2^n}, i = -1.$$

Then the probability that output port n does not receive a message from any of the input ports is given by

$$\left(1 - \frac{p}{2^n}\right) \prod_{i=0}^{n-1} \left(1 - \frac{p}{2^{n-i}}\right).$$

The probability that a message is received is then given by

$$P_r = 1 - \left(1 - \frac{p}{2^n}\right) \prod_{i=0}^{n-1} \left(1 - \frac{p}{2^{n-i}}\right).$$

Now consider the case when we have I/O messages also sharing the system links. Fig. 4 depicts this situation. We have a certain fraction of the messages arriving at each input port directed to the output port n. If these I/O messages are a small fraction of the whole message traffic, then we can still approximate the message distribution as being uniform. Then the probability that a message is received is given by

$$\left[1 - \left(1 - \frac{p(1+h)}{2^n}\right) \prod_{i=0}^{n-1} \left(1 - \frac{p(1+h)}{2^{n-i}}\right)\right].$$

In the above derivation, we assumed that the probabilities of the messages arriving at the switches are close to the probability of receiving a message at one of the inputs of a switch when no collisions occur in the network. However, collisions take place during message routing and this needs to be taken into consideration for a more rigorous analysis. If these collisions are assumed to cause the messages (except one) to be dropped, the probability of a message arriving at an input port of a switch will be some steady value $p'' < p/2$. This approximation hence gives a slightly optimistic estimate of the message completion rate. This is somewhat offset by the following pessimistic assumption. The probability P_r gives a worst-case estimate of the collision since the other output ports of a switch cannot receive messages from all the input ports. Hence, the average probability of a message being successfully routed to a destination in a circuit-switched system can be approximated by $P_r^{d/2}$ since a message on an average will have to be routed through $d/2$ switches. The actual difference in probabilities of routing a message successfully in the two cases now can be found by substituting the appropriate values of p in P_r. The difference in the two values tells us the penalty incurred due to sharing the links in the system. For example, when $h = 10\%$, $p = 1/10$, and $n = 6$, the

TABLE I
WORST CASE % INCREASE IN MESSAGE-DELIVERY TIME DUE TO SHARING LINKS

Message rate Inter-arrival time	% of I/O communication			
ms	5	10	15	20
10.0	0.4	0.75	1.09	2.01
5.0	0.74	1.53	1.94	2.94
2.5	1.79	3.15	3.89	5.77
1.25	3.14	6.76	9.05	12.64

probability that a generated message reaches its destination is 0.904 and 0.896 in the two cases, respectively. As can be seen from the numbers, the probabilities are not significantly different. A similar analysis has been recently reported in [17].

Another way of characterizing the network performance is to estimate the delay for sending a message. Table I gives the relative increase in delay for various message and I/O rates obtained through simulations. To see the worst-case effects, the simulations assumed that the I/O communication always collides with some message communication. Normally, the I/O communication collides with message communication only occasionally. The messages are not given any priority over the I/O communication in the simulations. The table reports the percentage increase in the message-delivery time due to sharing links. As can be seen from the table, even in the worst case, the degradation in performance is tolerable at moderate loads. When the I/O and messages do not collide and if the messages are given higher priority than the I/O communication, the performance degradation would be lower than that shown in Table I.

In systems where a single host serves the I/O needs of all the processors, the host acts as a *hot spot* when all the processors try to do an I/O simultaneously. In regular scientific problems, all the processors in fact try to do an I/O at about the same time creating a *hot spot* traffic to the host. As shown by [18], the effective bandwidth of the network in such a case is limited by the *hot spot* traffic. If I/O message rate is a fraction h of the message rate r, the total I/O message rate will be rhp. Asymptotically, the maximum value of the network throughput is then given by

$$B_{\text{eff}} = \frac{B}{ph},$$

where B is the bandwidth of the network without the *hot spot* and h is the fraction of the traffic that is directed towards the *hot spot* and p is the number of processors in the system. As can be seen, even if h is fixed, as the number of processors grows, the effective bandwidth of the system is reduced further.

The proposed architecture replaces the single host by a number of hosts. This effectively replaces one *hot spot* of the network by a number of *hot spots*. However, the traffic directed towards each *hot spot* is the I/O traffic of only a constant number ($c = 4$) of processors. When all the I/O requests of a processor can be serviced by the neighboring I-node, each host serves as a sink for I/O requests from c processors. Hence, the *hot spot* effect reduces the bandwidth of the system to

$$B_{\text{eff}} = \frac{B}{ch}$$

where c is the number of processors sharing an I-node. Hence, using multiple disks or multiple hosts, we can limit the performance reduction to a constant factor irrespective of the system size. This is a result of increasing the I/O processing capability along with the processing power of the system.

IV. EVALUATION OF DISK I/O SYSTEMS

The communication issues in the network and the need for balancing the I/O with the processing power of the system strongly motivate the use of multiple disk I/O systems. What is the best way to organize the different disks to get the required performance? This issue is investigated in this section through simulations. We use two different workloads for evaluating the different disk organizations: 1) file/transaction system workload and 2) scientific applications workload. These two workloads are expected to capture most of the requirements on an I/O system. The file system workload represents a multiuser environment with a lot of random file access and the second workload represents a single user environment with mostly sequential access to a few files.

With each disk I/O request, there are three main components of service time, namely, seek time, latency time, and read time. Seek time is the time spent in accessing the right track of the disk and latency time is the time taken for the required block to come under the read/write head. Read time is the time taken for reading the data from the disk to a buffer. Latency and seek times are overheads incurred in reading a block of data. The response time of a request is the sum of its service time and the waiting time in the queue.

A number of disk organizations are evaluated to see which is best suited to a hypercube system. We will use the terminology introduced in a related paper to explain the various organizations [19]. A synchronized system is one where a number of disks (m) are synchronized together to function as a single larger disk with m times the transfer rate of a single disk. The number of disks synchronized together is called the *degree of synchronization*. When the transfer time dominates the I/O service time, this organization yields high performance. Since all the disks are coupled together, each disk sees the same request rate and hence a balanced load on all the disks. The utilization of each disk is higher since each request has to be processed by all the disks. Hence, this organization is useful when large files need to be transferred.

A declustered system is one where each file is distributed on a number of disks such that different blocks of the same file can be accessed in parallel from different disks. The file is interleaved such that if block i is on disk j, then $i + 1$ is on $j + 1$ and so on. If the I/O request is for a single block, then that request is queued in the disk containing that block. If a multiple-block request arrives at the disk system, then the

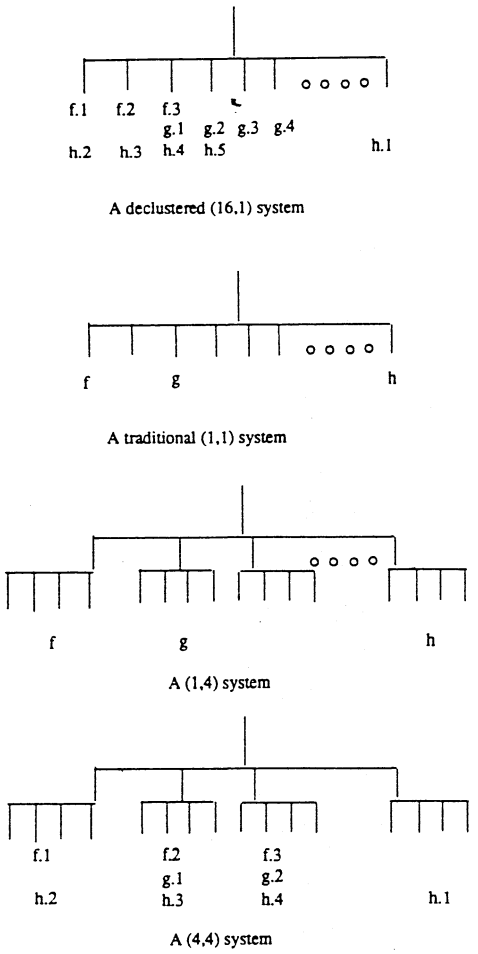


Fig. 5. Some of the disk organizations.

request is broken up into a number of single-block requests and it is queued at the corresponding disks. This results in concurrent access to the blocks. This organization also enables serving multiple single-block requests concurrently. By interleaving the data, we can keep the disk system balanced with respect to request rates. In this organization, the seek and latency penalties are paid for each block of transfer and hence the service time has effectively increased. The study in [7] indicates that the declustering approach trades-off service time effectively to reduce the variance in the queueing times of a traditional system to achieve high performance. In a nondeclustered system, each file is stored on only one disk. *Degree of declustering* of a disk system denotes the number of disks over which a file is declustered. For a traditional system, $dd = 1$ and for a fully declustered system, it is equal to the number of disks in the system.

Each disk organization can then be characterized by three numbers (dd, ds) and m where dd is the degree of declustering and ds is the degree of synchronization and m is the number of disks in the system. Fig. 5 shows different disk organizations.

We consider a 16-node hypercube system with 16 disks. All possible organizations with 16 disks were considered for evaluation. The I/O traffic is assumed to utilize the system links as explained in Section II. The system is assumed to contain four I/O nodes and the data transfer to and from the disk units is handled by the I/O nodes. The I/O nodes are

TABLE II
PARAMETERS OF THE SYSTEM

Parameter	Value
Max. latency time	16.6 ms
Seek time/track	0.04ms
Tracks per disk	1258
# of heads	14
# of sectors/track	52
# of bytes/sector	512
# of Disks	16
Deg. of Declustering	1,4,8,16
Deg. of Synchronization	1,2,4,8,16

interconnected among themselves and these links are used for I/O communication between two I/O nodes. Data are assumed to be transferred using a circuit-switch model. The latency time and transfer time for communication over the links are obtained through actual measurements over an Intel iPSC/2 hypercube. Since the (1,8) and (1,16) organizations have fewer than four independent units, there is no need for four I/O nodes. The I/O nodes are assumed to be present at locations 0, 7, 8, and 15 of the hypercube and in the (1,8) organization at 0 and 15 and in the (1,16) organization at 0. The disk characteristics are assumed to be similar to an RA81 disk [20]. The system parameters are shown in Table II. Each disk is assumed to use a SCAN approach of scheduling the requests, i.e., the head of the disk scans inwards starting from the outermost track serving any requests in its way till it reaches the innermost track and starts serving requests once again from the outermost pending request. This type of scheduling does some scan optimization and at the same time guarantees that no request is indefinitely postponed.

A. File/Transaction System Workload

The files in the system are assumed to be distributed randomly over the disks. In a declustered system, each file's blocks are distributed in a round-robin fashion among the disks. Each file is assumed to be 4 megabytes large and the system is assumed to have 800 such files. Consecutive blocks of a file are assumed to be located on contiguous tracks of the disk. In declustered systems, different blocks of a file that reside on the same disk are assumed to lie on contiguous tracks of the disk. Each I/O request specifies a file number and a block number within the file. Files are assumed to be accessed uniformly and similarly the blocks within a file are assumed to be uniformly accessed. Each simulation in the file system workload is carried out for 20 000 cycles such that a steady state is reached.

The file system workload is simulated as follows: the nodes in the hypercube generate messages at random instants of time. The number of messages generated at each instant is assumed to be uniformly distributed between 1 and 4. The time between two instants of message generation is assumed to be

exponentially distributed and we call this parameter message-generation time. The message-generation time is varied from 1.25 ms to 10.0 ms. A certain fraction of the time, I/O messages are also generated along with the regular communication messages. This fraction is varied from 5 to 20%, which we will denote as the *I/O frequency*. Actual fraction of I/O communication (in terms of bytes) is higher because of larger data sizes used in I/O transfer, 4 kilobyte blocks compared to 1 kilobyte processor-to-processor messages. In most workloads, the typical I/O needs are expected to be within this range. The I/O requests are divided into three different classes. The first class 1) "broadcast," generates a request for a block of data that is required by all the nodes of the system. The second class 2) "multiple" generates multiple requests to the same file. A number of nodes uniformly distributed over 2 and 16 each ask for a different block of data from the same file. The third class 3) "single" generates an I/O request for a single block of data. The percentage of requests in classes 1, 2, and 3 are 5, 35, and 60%. This is expected to be the typical workload in a hypercube system. The I/O in a hypercube can be divided into three phases: 1) program loading where all the nodes request the same information, 2) data distribution where each node requests for some data that it needs to work on, 3) writing results back where all the nodes write the obtained results back to the disk. These three phases of I/O load are modeled by the above three classes of I/O requests. The block size of the I/O system was fixed at 4 kilobytes.

Average response time of the I/O system is used as a measure of performance. The response time of a request is considered to be the time between the instant the processor generates an I/O message and the instant it receives it. In case of multiple and broadcast classes, the I/O transfer is considered complete only when all the nodes receive the required data. This measure includes the time taken for transferring the data to and from the disks over the communication links. Network congestion as a result of placing the disks at too few locations was one of the issues to be investigated.

Fig. 6 shows the average response times as the inter-message generation time is varied at a 10% I/O request rate. Fig. 7 shows the effect of varying the I/O request rate, keeping the inter-message generation time fixed at 5.0 ms. The (1,16), (1,8), and (2,8) organizations are saturated when the message-generation time is less than 2.5 ms. As noted from Fig. 7 also, the (1,16) organization is very sensitive to the load on the system. The (1,1) and (16,1) organizations seem to be very insensitive to the frequency of io-requests within the considered range.

Fig. 8 shows the average time in the network as a part of the I/O transfer. By comparing Figs. 7 and 8, the network congestion is observed to be not a factor in the performance of these systems with the considered workload and the system sizes. The amount of time spent within the network does not vary significantly and is not a major contributor to the response time of the I/O request. The communication time consists of two parts, the disk-to-I/O buffer transfer time and the I/O buffer-to-processor transfer time. The synchronized systems utilize the communication links in parallel to transfer each block of data and hence the disk-to-buffer communication time is lower. This results in the observed trend in the communication times. The I/O communication is low enough that we do not observe the *hot spot* effects of organizing all the disks as a single unit as in (1,16).

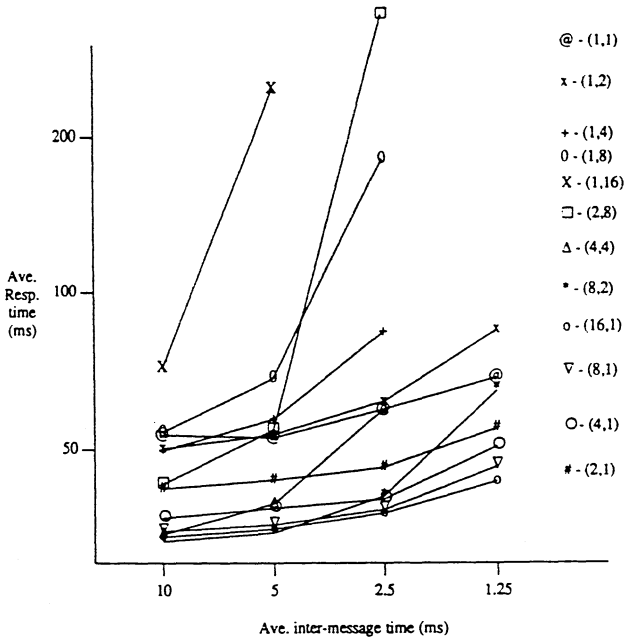


Fig. 6. Average response time versus message rate.

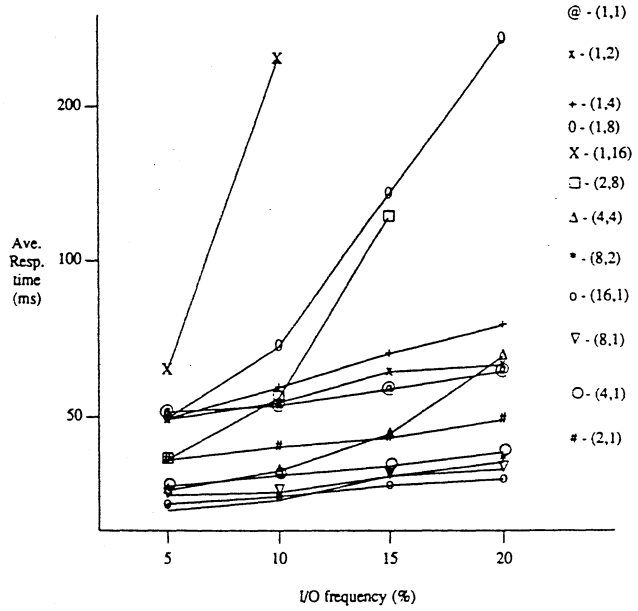


Fig. 7. Average response time versus I/O rate.

The main difference in the workload used here compared to the one used in the other study [19] is that the multiple requests generated simultaneously for a single file are assumed to be for contiguous blocks of data in the earlier evaluation. This assumption biased the results in favor of the synchronized systems. In the earlier study, the requests to multiple blocks of a file are assumed to be contiguous and generated by a single process. Hence, rotational penalty was paid only once in accessing multiple blocks in synchronized systems. However, in this workload, the requests for multiple blocks of a file are generated by different processes and hence we treated them as separate requests. This assumption made a considerable differ-

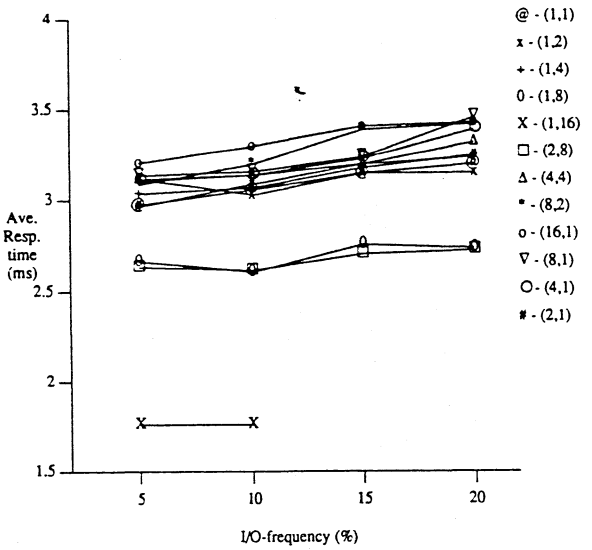


Fig. 8. "Network" time of I/O response versus I/O rate.

ence in the results of synchronized systems in the two studies. This shows that the results obtained for a single processor environment cannot be easily extended to a multiprocessor environment since the request patterns are considerably different. The difference in the results of these two studies arises due to the difference in the assumptions about serving multiple-block requests at synchronized systems. When the best-case scenario is assumed, i.e., all the blocks in a multiple-block request are located on contiguous locations of the disk as in the earlier study [19], the synchronized systems perform better at lower request rates. In the scenario considered here, the blocks are assumed to be accessed in random order. As a consequence of paying multiple latency penalties in serving such multiple-block requests, the synchronized systems perform worse than the declustered systems even at low request rates. This points out to the importance of request scheduling at the disks. The other difference in this study is the extra communication cost incurred in sending the message from the I/O nodes to the processors.

B. Scientific Applications Workload

The second workload considers the situation where large scientific applications are executed in a single-user environment. Since the scientific computations are structured and regular, it is possible to alleviate some of the I/O needs of a problem through prefetching. However, if the data processing speeds and data reading (from disk) speeds are considerably different, the amount of data that needs to be prefetched could be considerably larger than what the buffers could hold. Hence, even with buffering there is a need to improve the basic data-reading time from the disks.

We consider three different matrix problems, matrix-matrix multiplication, matrix-vector multiplication, and the FFT, for our evaluation. These three problems have different I/O requirements. The ratio of the total number of computations to the total number of I/O operations is $O(\sqrt{n})$ for matrix-matrix multiplication, $O(\log n)$ for the FFT, and $O(1)$ for matrix-vector multiplication. It is expected that these three applications give a wide range of I/O requirements such that observations can be made regarding the performance of the different organizations. We use a hypercube multiprocessor model to simulate parallel processing of the data. The I/O patterns obtained in such a model are very sensitive to the assumptions made about the relative sizes of memory and the problem. Here we assume that, for each problem, the size of all the available memory space on the processors is large enough to hold the required data for solving the problem. This assumption leads us to obtaining I/O traffic patterns of the following form: every processor requests for a piece of data, solves the problem in a cooperative way, and then writes the results back to the I/O system. This I/O traffic pattern is near-synchronous. Since we did not have actual I/O traces, we had to restrict our attention to such a model. This assumption also frees us from further assumptions about how fast the memory size should grow as a function of processor speed. An evaluation of IBM RP3 architecture regarding the I/O issues, for seismic applications, can be found in [21].

The I/O subsystem consisted of four I/O nodes connected to the hypercube multiprocessor system as described in [11]. Different disk organizations as discussed earlier were considered for evaluation. The simulations were carried out with the same processor system but different I/O subsystems. Since the algorithms are assumed to be implemented exploiting all the features of the system, any differences in performance are mainly due to the differences in the I/O subsystem. The data are transferred over single links from I/O nodes to the processors in the system. However, the data are assumed to be transferred over multiple links (where available) from the disk to I/O buffers. The time taken to solve the given problem was taken to be the performance measure. We simulated the problem with different processor speeds. Starting with a base speed of 1 (roughly equivalent to 0.5 MFlops), the processor speed was varied from 0.1 to 10000 relative to the base speed. The problem size was fixed through simulations. Since the data size was assumed to be small enough to fit into the memory of the system, we did not have to make further assumptions about increases in memory size as the processor performance is increased. All the processors are assumed to obtain the required data before beginning the processing phase. Blocked storage is used for declustered systems. Since the files are stored as single entities in nondeclustered systems, row/column major form of storage is assumed in such systems.

In application 1, multiplying two 1024×1024 matrices was simulated. The matrices A and B are distributed equally among different processors. To generate matrix $C = AB$, each processor multiplies its part of A with B. Hence, the different pieces of matrix B are circulated among the processors during the problem execution. For example, in a nondeclustered system each processor multiplies its rows of A with its columns of B and obtains a new set of columns of B from another processor for further processing and so on till the matrix C is generated. Only matrix C is written back. Fig. 9 shows the performance of various machines executing matrix multiplication.

In application 2, a two-dimensional FFT algorithm on 1024×1024 points was simulated. The data are distributed

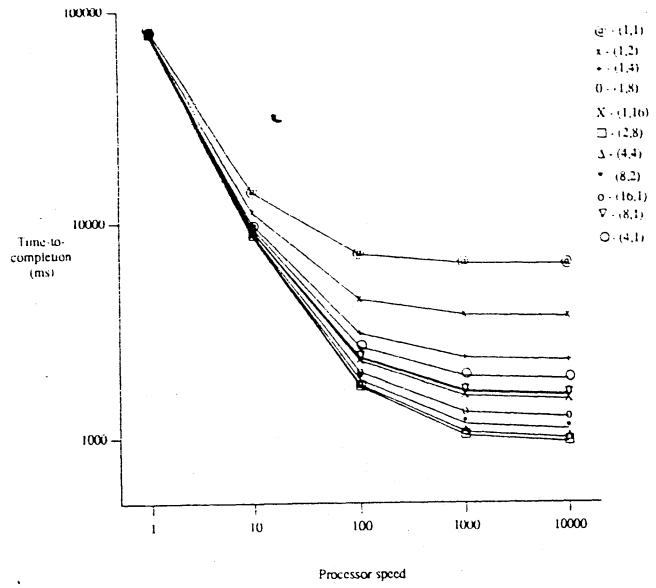


Fig. 9. Performance of different organizations with matrix multiplication.

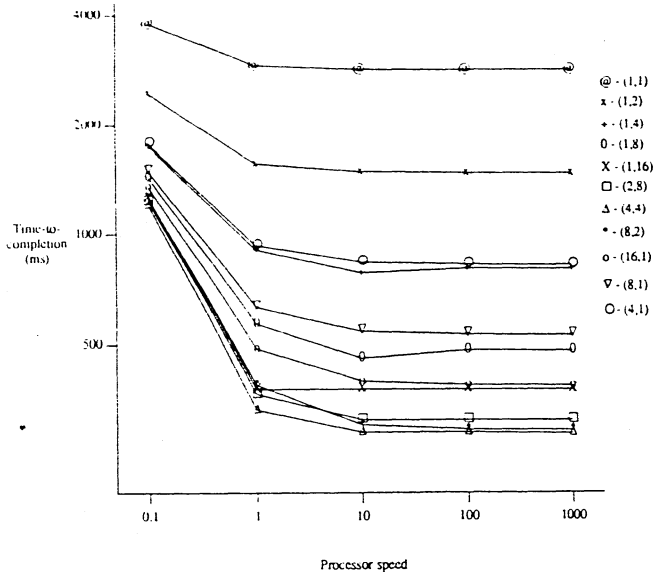


Fig. 11. Performance of matrix vector multiplication.

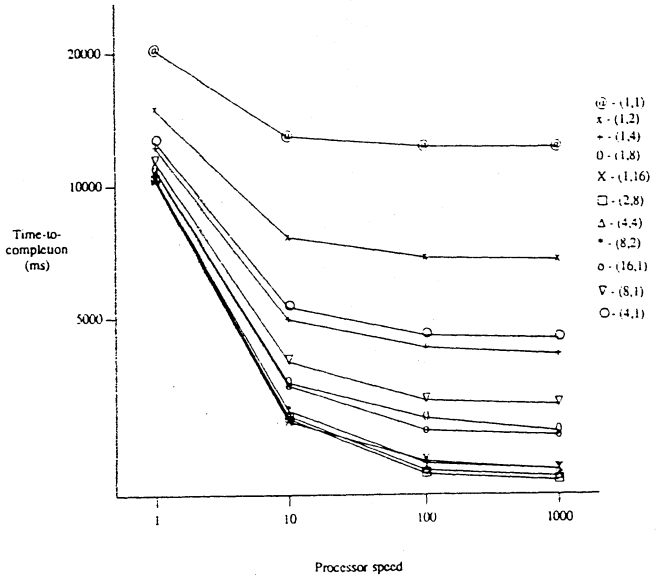


Fig. 10. Performance of FFT.

to different processors, reorganized among the processors, if necessary (in declustered systems), to obtain a row-wise partition of the data. Each processor computes a one-dimensional FFT of each of its rows. The matrix is transposed to get a column-wise partitioning for the second phase. Each processor computes a one-dimensional FFT of each of its columns to obtain the two-dimensional FFT of the data. The data are reorganized into blocked-form or row-major form and written back to the disks. Fig. 10 shows the performance of different organizations executing FFT.

In application 3, multiplying a 1024 × 1024 matrix with a 1024 vector was simulated. Each processor receives part of the matrix and all of the vector. The data are organized into row-major order if necessary. Each processor computes its part of the matrix-vector product and writes back the resultant vector to the disks. Fig. 11 shows the performance of different organizations executing matrix-vector multiplication.

Figs. 9–11 show the performance of various machines as a function of processor speed for the three problems matrix multiplication, FFT, and matrix-vector multiplication, respectively. The following observations can be made from the figures:

1) Matrix multiplication is compute bound at processor speeds up to 1000, FFT up to 100, and matrix-vector multiplication up to 1. As expected, these three problems exhibit different balances between computations and I/O. Beyond these points, the problems are I/O bound. The differences in the I/O performance can be noted when the problem is I/O bound. The differences in performance are due to the differences in I/O subsystem since the multiprocessor system is unchanged in all the organizations. The organizations that offer higher parallelism for reading data from a single file, i.e., with higher $ds*dd$, perform better than organizations with lower parallelism. Even when the problem is compute bound, although relatively minor, the differences in performance can be noticed.

2) The traditional (1,1) organization is seen to perform very poorly compared to other organizations. This shows that putting together a number of disks without increasing the concurrency of access to a file is not a good choice.

3) The time-to-completion does not scale linearly with the parallelism available ($ds*dd$) for reading data. For example, time-to-completion for FFT is 12134.80 for (1,1), 2764.35 for (1,8), and 2273.36 for (1,16) at a processor speed of 1000. The main reason for this is the transfer time over the system links from the I/O node buffers to the processor memories. Since data are transferred over single links from the I/O nodes to the processors, the I/O bottleneck now has become a data-transfer bottleneck. This implies that in the future the data transfer speeds also have to increase significantly to solve the I/O problem.

4) The differences in performance of the organizations with $dd*ds = 16$ do not differ considerably to conclude about their relative performances. However, the simulations clearly show the need for parallelism in I/O.

Similar results are obtained in a related paper for a shared-memory environment [19]. The effect of transferring the data

over the single system links is visible in this study and we noted the fact that the I/O speed did not scale with the parallelism because of the communication bottleneck. In the other study, we only considered transferring the data to the shared memory region and since this was done on parallel communication links, we did not observe a similar communication bottleneck. Otherwise, the results regarding the relative performance of the different I/O systems are similar in the two cases.

The synchronized systems cannot be scaled easily. Consider building a system with larger number of disks. The disks cannot be synchronized beyond a certain number, maybe 8 or 16. Beyond that point, some form of declustering has to be considered to improve the parallelism. The results in this section tell us what is a good way to organize a small number of disks. The available parallelism in serving a request plays a dominant role in performance. Declustered-synchronized organizations may be a good choice for scientific applications in larger systems since they offer maximal parallelism in serving single-block requests and multiple-block requests.

V. Matrix Distribution

The earlier section showed that some degree of declustering is needed to improve the I/O performance considerably. What is the best way to store the data on the disks? This question cannot be answered without knowledge of the details of the data and how it is used. At a system level, a uniform approach of distributing the data by the bytes among the disks is the only feasible approach. Matrices are used extensively in scientific applications and most matrix applications can be characterized by a few basic operations. Hence, it is possible to find some simple ways of storing a matrix on the different disks such that the I/O requirements of the application are efficiently satisfied. We present below a method to store the matrices on the disks.

When the processor needs to perform an I/O operation, it sends the request to the neighboring I-node. If the neighboring I-node can satisfy the request, it serves the I/O. Most scientific problems can be decomposed in a regular fashion where the interaction between different pieces of the data is minimal. Matrix arithmetic and some image processing applications are such examples. In such applications, each processor executes a set of operations on the piece of data it owns and does not extensively use the data owned by the other processors. In such cases, the problem is mapped onto the hypercube system assuming that each processor has an infinite amount of memory. The problem mapping involves several considerations, available parallelism, minimization of messages, etc. Once the problem mapping is complete, the data are mapped from the nodes to the neighboring I-nodes. By this approach, we can minimize the communication between the nodes in the system and at the same time make the data available to the nodes from their neighboring I-nodes. This process of mapping data can be seen to be similar to the approach taken for register allocation by a compiler where variables are first mapped onto an infinite number of registers enforcing the flow order restrictions and later remapping them onto a finite set of registers.

The advantages of this approach are 1) message communication is minimized during the execution of the problem, 2) an explicit I/O request is most likely served by the neighboring I-node. In problems where pieces of data are not used by more than one processor, the above discussed method of data partitioning based on load balancing, message minimization yields best results. In such cases, it is to be left to the user to do the necessary I/O minimization. The disadvantage of this approach is that the data mapping needs to be done for each specific problem. The user has to somehow remember how the data are stored among the various disks for each piece of data. Moreover, if a number of operations are performed on the data, it is likely that the data may be required to be mapped onto the I-nodes in different forms. A uniform approach is required for distributing the data onto various I-nodes in such cases. The aim of this section is to suggest an approach to distribute data onto I-nodes when the data may be used in different configurations and when each processor may need several pieces of data to complete the execution. Matrix multiplication is a good example where each processor operates on several pieces of data and the data are required to be in different configurations (in rows or columns) for an efficient solution.

A second method of data mapping is based on using size-independent algorithm design. In this method, each problem or algorithm is designed such that it uses the minimum amount of space for execution. For this approach, algorithms are written such that they execute on small blocks of data. When a larger problem is to be solved, the block algorithm is repetitively used to produce necessary results. Using such an approach, a given piece of data can be mapped onto the system and then several algorithms if need to operate on the same data could still operate at a reasonable efficiency. A good reason for using a block approach for storing data on disks is illustrated by the following: consider a matrix A that is needed in a premultiplication and a postmultiplication. If A is stored in a row-major fashion among the I-nodes, then premultiplication can be done efficiently. But, postmultiplication of A would either require that each processor read data from all the I-nodes, or a transpose of the row-major ordered matrix, an expensive operation. Similarly, we can argue that storing the matrix in a column-major order also leads to an inefficient operations on the matrix. It has been shown that when the data are stored in the form of matrices, storing the matrix as a number of rectangular blocks yields best efficiency in a virtual memory storage [22], and also in hierarchical memory organizations of shared-memory multiprocessors [23]. Algorithms are written such that they can operate on the blocks and combine the results if necessary to produce the results on larger pieces of data. This approach is more general than the earlier problem specific approach and would be useful if a number of different operations are carried out on the same piece of data. However, there remains one issue that needs to be resolved, how should the various blocks be distributed among the I-nodes?

We propose a method of storing different pieces of data on the I-nodes by using a combination of both the strategies above. The data distribution is based on the following assumptions: 1) data stored on disks be read only once from the disk

even if required by a number of processors; one processor will do the I/O read and distribute this data to other nodes through messages. 2) The system size is known and all the processors are available for solving a particular problem. This assumption is a little strong for various reasons. The problem size may be such that the optimum number of processors used for that application may be different from the system size. In a multiuser environment, it may not always be possible to get hold of the complete system to solve a problem. The mapping problem is strongly related to the system size and without a knowledge of the system size it is impossible to determine a good mapping of data. Moreover, when possible we would like to extract maximum parallelism from the system.

We illustrate the data distribution approach with a two-dimensional matrix. But the ideas can be extended to higher dimensions. The data distribution involves the following steps: 1) divide the data into blocks, 2) map the blocks onto the nodes of the system to minimize communication, and 3) then move the mapped data onto the neighboring I-nodes for the later access.

Based on the arguments of minimizing the disk reads, we would like the data to be ordered in a blocked fashion as explained above. Consider a four-dimensional hypercube. Since the system has 16 processors, the matrix needs to be divided into 16 pieces. Since different pieces of data will have different sizes, the blocks of data will also have different sizes. Hence, the disk system should allow variable block storage. If the disk system has a fixed block size then the blocks of data can be stored as a number of blocks. The data blocks should be such that they can be individually operated on, large enough that disk accesses are efficient, and small enough to fit in the available memory at the nodes.

Consider dividing matrices $A_{n\times m}$ and $B_{m\times k}$ into blocks. If the number of processors in the system is a square number 2^{2l}, then we can map a mesh of size $2^l \times 2^l$ onto the system. Then the blocks of the matrices A and B will be of the sizes $n_1 \times m_1$ and $m_1 \times k_1$ where $n_1 = n/2^l$, $m_1 = m/2^l$ and $k_1 = k/2^l$. If matrices A and B are such that we could multiply AB, then the blocks are also of sizes such that the blocks can be multiplied. But when the system does not have a square number of processors, can we guarantee such a property? To satisfy this condition, we suggest the following: consider a system with 2^{2l-1} number of processors. Assume that we have 2^{2l} number of processors and divide the matrices into $2^l \times 2^l$ number of blocks. Now the blocks have the desired property as described above. Since we do not have as many processors we have to map two blocks onto each processor. If we were to divide the matrix into a number of blocks such that each processor has only one block, then the blocks will not be of the right sizes to carry out multiplication and problem solving with such blocks will be very difficult.

Once the data are divided into a number of blocks, what is the best way to distribute these blocks to different processors in the system? The answer to this depends on the operations to be performed on data. We suggest a mapping such as shown in Fig. 12. The numbers in the blocks correspond to the processor addresses in the hypercube. The motivation for proposing this distribution follows. For algorithms such as matrix multiplication, there exist efficient block-implementations. But consider a problem such as two-dimensional FFT for which there is no efficient block implementation. In such cases, we would like to reorder the data into either row or column major order. To redistribute the data blocks into rows or columns, the connectivity among the processors holding these blocks should be maximum. This implies that processors holding the first four data blocks should be maximally connected. A similar property has to be satisfied for other rows of blocks. Using a similar argument, we require that the processors holding the columns of blocks be maximally connected. There exist several such mappings as shown in Fig. 12. Of these mappings, we choose the gray-code mapping of Fig. 12(b) as a possible candidate for data distribution since it is easily understood. If the data are then distributed to various nodes in such a fashion, it is seen that consecutive blocks of data among rows and columns are held by neighbors in the hypercube.

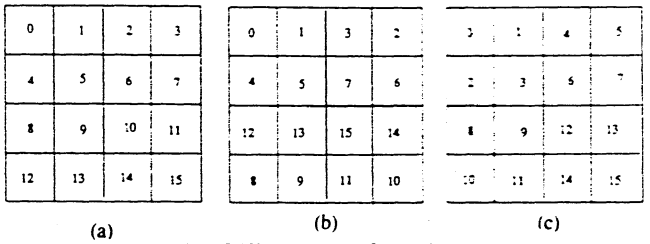


Fig. 12. Different ways of mapping data.

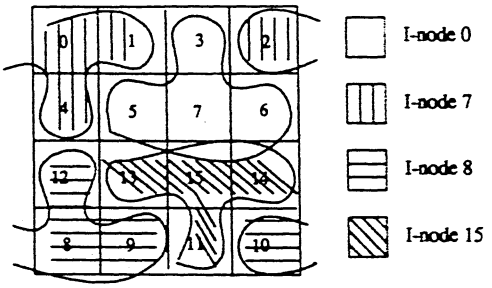


Fig. 13. Data distribution at I-nodes.

The last step of the data distribution consists of moving these mapped blocks to their neighboring I-nodes. Then the distribution of data at each I-nodes is as shown in Fig. 12 for our example.

For an n-dimensional matrix, we can map the number of processors in the system into an n-dimensional mesh and map the pieces of data into the corresponding nodes in the system. A vector can be divided along its length to be stored among the different disks. However, it is to be noted that making too many small pieces of data may not improve the response time.

The advantages of blocking can be clearly seen when we need to do a matrix transposition. If the data are stored in row-major order or column-major order, then each processor needs to communicate with every other processor if the memory at each node is limited. But using a distribution such as the one in Fig. 13, the number of communication steps are reduced considerably. In fact, the transposed matrix is easily available by a suitable renaming of processor numbers. Observing Fig. 12(b), the transposed matrix is available by cyclically shifting the processor number by two bits.

VI. Conclusion

In this paper, we looked at several issues concerning the I/O system performance in a hypercube. We presented a systematic method to connect the I/O processors to the nodes in the multiprocessor system. We analyzed the communication traffic in the network to show that I/O and message communication could share some of the system links without excessive performance penalties. Several disk organizations are evaluated and it was shown that different organizations perform better in different workloads. It was observed that parallelism in serving an I/O request plays a dominant role in the scientific workload. We also presented a way of declustering matrices on the disks such that they can be accessed efficiently for matrix arithmetic operations.

References

[1] J. P. Hayes *et al.*, "Architecture of a hypercube supercomputer," *Proc. ICPP,* pp. 653–660, 1986.

[2] "Intel IPSC system product summary," Intel, OR.

[3] J. Tuazon, J. Peterson, M. Pniel, and D. Liberman, "Caltech/JPL mark II hypercube concurrent processor," *Proc. ICPP,* pp. 666–673, 1985.

[4] G. C. Fox and S. W. Otto, "Algorithms for concurrent processors," *Phys. Today,* pp. 13–20, May 1984.

[5] H. T. Kung, "Memory requirements for balanced computer architectures," in *Proc. 13th Annu. Int. Symp. Comput. Architecture,* 1986, pp. 49–54.

[6] S. Cannon, "Concurrent file system—Making highly parallel mass storage transparent," in *Proc. Supercomputing '89,* St. Petersberg, FL, Apr.–May 1989.

[7] M. Livny, S. Khoshafian, and H. Boral, "Multi-disk management algorithms," in *Proc. ACM SIGMETRICS,* May 1987, pp. 69–77.

[8] M. Y. Kim, "Synchronized disk interleaving," *IEEE Trans. Comput.,* vol. C-35, pp. 978–988, Nov. 1986.

[9] Kai Hwang and J. Ghosh, "Hypernet: A communication-efficient architecture for constructing massively parallel computers," *IEEE Trans. Comput.,* vol. C-36, pp. 1450–1466, Dec. 1987.

[10] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes.* Cambridge, MA: M.I.T. Press, 1984.

[11] A. L. Narasimha Reddy, P. Banerjee, and S. G. Abraham, "I/O embedding in hypercubes," in *Proc. Int. Conf. Parallel Process.,* 1988.

[12] M. Livingston and Q. F. Stout, "Distributing resources in hypercube computers," in *Proc. 3rd Conf. Hypercube Concurrent Comp. Apps.,* Jan. 1988.

[13] H. Sullivan and T. R. Bashkow, "A large scale homogeneous machine," in *Proc. 4th Annu. Symp. Comput. Architecture,* 1977, pp. 105–114.

[14] J. H. Patel, "Analysis of multiprocessors with private cache memories," *IEEE Trans. Comput.,* vol. C-31, pp. 296–304, Apr. 1982.

[15] C. P. Kruskal and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Trans. Comput.,* vol. C-32, pp. 1091–1098, Dec. 1983.

[16] D. M. Dias and R. Jump, "Analysis and simulation of buffered delta networks," *IEEE Trans. Comput.,* vol. C-30, pp. 273–282, Apr. 1981.

[17] S. Abraham and K. Padmanabhan, "Performance of the direct binary *n*-cube network for multiprocessors," *IEEE Trans. Comput.,* vol. 38, pp. 1000–1011, July 1989.

[18] G. F. Pfister and V. A. Norton, "Hot spot contention and combining in multistage interconnection networks," *IEEE Trans. Comput.,* vol. C-34, pp. 943–948, Oct. 1985.

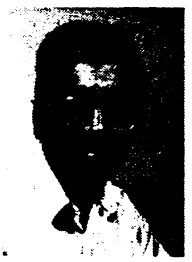
[19] A. L. N. Reddy and P. Banerjee, "An evaluation of multiple-disk I/O systems," *IEEE Trans. Comput.,* vol. 38, pp. 1680–1690, Dec. 1989.

[20] *RA81 Disk Drive User Guide.* Digital Equipment Corp., 1982.

[21] S. Sivaramakrishnan, "Evaluation of logical external memory architectures for multiprocessor systems," Tech. Rep. TR-88-32, Dep. Comp. Sci., Univ. Texas, Austin, Sept. 1988.

[22] A. C. McKellar and E. G. Coffman, "Organizing matrices and matrix operations for paged memory systems," *Commun. ACM,* vol. 12, no. 3, pp. 153–165, Mar. 1969.

[23] K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "The impact of hierarchical memory systems on linear algebra algorithm design," CSRD Rep. No. 625, Sept. 1987.



A. L. Narasimha Reddy received his B.Tech degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1985 and the M.S. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign, in 1987.

Currently he is pursuing the Ph.D degree in electrical and computer engineering at the University of Illinois, Urbana-Champaign. He is the recipient of an IBM Graduate Fellowship. His research interests are in parallel processing, computer architecture, and fault tolerance.

Prithviraj Banerjee (S'82–M'84), for a photograph and biography, see p. 106 of the January 1990 issue of this Transactions.